Chapter 2 Approaches to Software Design

Software design emerged during the 1960's as a topic for consideration, practice, and research. Over the past three decades numerous software design methods have appeared, some finding their way into use, others contributing interesting ideas. Section 2.1, below, discusses the objectives of software design methods in general and some of the more notable software design methods in particular. Researchers have investigated automated approaches to assist designers in applying some of the more popular software design methods. Section 2.2 outlines and critiques a number of such approaches to automating software design methods, both at the architectural level and at the detailed-design level. A possible alternative to automation-assisted methods for software design, known as automatic programming, attempts to replace the software designer by generating executable code directly from a specification or by interacting directly with a user. Section 2.3 identifies several examples of automatic programming and critiques the approach. The chapter closes with a summary of findings.

2.1 Software Design Methods

Designers face a problem fundamentally different from the problem faced by natural scientists. A scientist examines nature in an effort to discern cause and effect relationships and to describe those relationships in the form of mathematical equations and scientific laws that enable us to predict the outcome of various physical situations. In short, a natural scientist investigates what is, and why. A designer, on the other hand, proposes what ought to be, and how.

This essential difference between natural science and design led Herbert Simon to include design within the category of disciplines that he dubbed the sciences of the artificial. [Simon81] According to Simon, "[d]esign...is concerned with how things ought to be, with devising artifacts to attain goals." [Simon81, p. 133] Four other, similar, views of design were reported by Peter Freeman [Freeman80] in a survey he conducted: 1) design is an imaginative jump from present facts to future possibilities, 2) design is finding the right components of a structure, 3) design is decision-making in the face of uncertainty with high penalties for error, and 4) design is simulating, iteratively, a proposed solution until one is confident about the outcome. In order to provide automated assistance to a human designer, researchers must develop a good understanding of a designer's objectives.

2.1.1 Objectives of Software Design Methods

One objective of design, overlapping to a degree with analysis, is to discover the structure of a problem. Within the realm of software this objective might be fulfilled by reviewing the informal software requirements specification and then by analyzing the requirements using some systematic method. A second objective of design is to create an outline, or architecture, of a solution for a problem. For software design, this objective might be met by describing a set of software components and the relationships among them in enough detail that further, detailed design and then coding can be performed on each component. A third objective of design is to evaluate the results of proposed architectures against the stated goals (i.e., the requirements). Software designers often fail to achieve this objective. Typically, no evaluation of a design occurs until system testing. Design flaws discovered during system tests can be quite costly to repair.

To meet these design objectives, a designer usually engages in a number of intellectual activities. [Freeman80] One such design activity might be called *operationalization*. Operationalization entails improving the informal requirements by removing ambiguities, by reconciling inconsistencies, and by adding missing information. The designer's job must start with operationalization because later design activities depend upon the system requirements. Another design activity involves *abstraction*. Here the designer generalizes about particular properties of the problem or of a possible solution; certain details are set aside at critical moments so that the designer can concentrate on a specific issue. Associated with abstraction, a designer employs *elaboration* to move down a hierarchy of levels of abstraction so that essential details can be provided at an appropriate time. Probably the most important intellectual act during design involves *verification*. A designer must verify that a proposed solution meets the requirements, any imposed standards, and any extant constraints. A designer must also be able to verify the performance characteristics of a proposed solution.

The essence of design, as embodied by the four intellectual activities of operationalization, abstraction, elaboration, and verification, is *decision-making*. Unfortunately, the record reveals that designers do not always make sound decisions.

> Experience with large software systems shows that over half of the defects found after product release are traceable to errors in early product design. Furthermore, more than half the software life-cycle costs involve detecting and correcting design flaws. [Beregi84, p. 4]

To ameliorate these problems researchers have focused on the development of design methods. Several design methods for distributed and real-time systems are discussed below, but for now consider, in general, how a design method can help. A design method specifies: 1) what decisions a designer must make, 2) how those decisions should be made, and 3) in what order they should be made. [Freeman80] A design method, then, provides the intellectual roadmap that enables a designer to refine requirements successfully, to apply abstraction and elaboration correctly, and to achieve design verification. Design methods aim to improve the ability of software designers to produce designs of reasonable quality on a repeatable basis.

Software developers, generally, design and implement software to fulfill a set of functional requirements and nonfunctional requirements. Functional requirements express the necessary logical characteristics of a correct solution. Nonfunctional requirements describe other operational constraints, such as performance, reliability, and specific target hardware. For real-time systems, the nonfunctional requirements take on an added importance. For so-called soft real-time (SRT) systems (sometimes referred to as interactive systems) the performance requirements might indicate a performance target given a specified load on the system. For so-called hard real-time (HRT) systems (sometimes referred to as reactive systems) the performance requirements can form a three-level hierarchy: 1) those that must be met for correct system function, 2) those that are soft (in the sense formerly discussed for SRT systems), and 3) those that have more lenient time constraints (usually called background functions). For real-time software, then, the performance requirements take on a functional flavor because a system that fails to meet the stated performance constraints is considered to be: 1) functionally degraded for soft real-time requirements and 2) functionally incorrect for hard real-time requirements.

Despite the increased emphasis on nonfunctional requirements, designers of real-time software aim to achieve the same, three general goals as designers of any other software: 1) understandability, 2) functional correctness, and 3) performance sufficiency. To achieve understandability the software designer must meet four sub-goals. First, the designer must ensure complete, consistent, and unambiguous functional requirements. Software requirements documents typically consist mostly of natural language descriptions augmented with some formal specifications that are generally applied unevenly. The designer must seek to improve the rigor of the specification, to fill the gaps, and to resolve contradictions. Without such efforts the designer cannot achieve an understanding of the problem sufficient to propose and evaluate solutions. The remaining sub-goals relate directly to design.

The designer must provide a clear structuring of the system into design elements, such as tasks and information hiding modules. Then the designer must specify the behavior of these elements. Finally, the designer must establish traceability between the structure and specification of the design and the software requirements. As a result of achieving these sub-goals, the designer produces an understandable, but unverified, design.

Next, the designer must work to ensure the functional correctness of the design at both the component and system levels. At the component level, the designer should specify partial correctness criteria for each sequentially executing path. Such paths typically include the program flow of control (one for each task in a concurrent design) and the services provided by each information hiding module. In general, the designer must specify preconditions and post-conditions for each design component such that if the preconditions hold on entry to the component, then the post-conditions will hold upon exit from the component. These specifications enable component designers and coders to understand precisely what their component must achieve, as well as to understand what should be provided to and expected from other components. Such specifications can also serve as a foundation for unit and integration testing during implementation of the design.

At the system level, designers of concurrent software have two concerns regarding functional correctness. One concern involves ensuring the absence of undesirable properties, such as deadlock, livelock, unfairness, failure, and unreachable states, that can occur in concurrent designs. [Karam91, Liu90, Levi90, Xu93] The designer must also establish that a concurrent design exhibits certain desirable properties, such as proper synchronization among communicating tasks, mutually exclusive access to shared resources, bounded behavior, and conservation of system resources. [Dillon90, Murata84, Murata89, Willson90, Zave86]

As a remaining concern, the designer must meet the performance constraints for the system. [Lien92, Liu90, Natarajan92, Levi90, Xu93] An initial complication arises when the requirements specification includes incomplete or inconsistent timing constraints. So the designer must first ensure proper specification of the system performance constraints. After attaining proper understanding of a system's timing requirements, the designer's major performance concern, for hard real-time systems, becomes ensuring schedulability of the software design under worst-case assumptions.

In summary, designers of software aim to create understandable designs that can guide implementation and provide traceability from the requirements specification. When the designs include concurrency, a number of implicit functional requirements must also be addressed. Designers of real-time systems must be concerned also about specific performance characteristics, such as maximal performance under a sustained load for interactive systems and worst-case performance under transient loads for reactive systems. Software design methods exist to help designers achieve these objectives.

2.1.2 Methods for Producing Sequential Designs

Two notable methods exist for producing sequential designs from flow diagrams. In the first method, a designer produces data flow diagrams and then transforms them into structure charts. In the second method, a designer constructs data/control flow diagrams and then generates structure charts. Other methods for producing sequential designs exist but are not described here. [Ingevaldsson79, Jackson75, Parnas72, Parnas76, Stay76]

2.1.2.1 Structured Analysis and Structured Design

One effective analysis and design method combines Structured Analysis [Demarco78] with Structured Design [Yourdon79]. Structured Analysis (SA) provides a simple, yet effective, method for representing and comprehending a range of data processing systems using a process, or transform-oriented, view of a software problem. Structured Analysis models a system as a hierarchical set of data flow diagrams (DFDs) depicting a series of steps that transform incoming data into outgoing data. Given a DFD specification for a system, a software designer must create a plan for a program that satisfies the specification and that can be coded by programmers. Structured Design (SD) provides a method for creating such a plan for sequential programs. [Yourdon79] Structured Design identifies desirable properties of a sequential, program design, provides a set of heuristics for transforming a SA specification into such a design, and also defines a notation, structure charts, for representing the design. Although Structured Analysis and Structured Design provide useful notations and approaches for designing many standard, software applications that admit sequential solutions, a wider range of software problems, including real-time systems and concurrent processing, can benefit from additional techniques. For this reason, researchers have developed real-time variants of Structured Analysis and Design.

2.1.2.2 Real-Time Structured Analysis and Structured Design

Real-Time Structured Analysis (RTSA) augments SA with additional semantics and notations to model events and control. [Ward85, Hatley86] The primary improvements made by RTSA include: control transformations and event flows. Control transformations, represented by circles enclosed in dashed lines, encapsulate state-transition diagrams (STDs) that order asynchronous events flowing into a system to control the processing and outputs of the system. In place of pseudo-code, a STD supplements each control transformation in a RTSA specification by depicting the control functions of the transformation. All inputs to and outputs from RTSA control transformations take the form of event flows, represented by dashed, directed arcs. Event flows can be further classified as signals, triggers or enable/disable switches. Signals represent events flowing into a control transformation. Triggers cause a data transformation to be activated for one execution of the processing represented by the transformation. Enable/disable events turn a data transformation on or off. Once enabled, the processing within a data transformation continues until disabled.

Given a RTSA diagram, a designer can use structured design heuristics to generate a structure chart for a sequential design, much as described previously. As an alternative to a sequential design, the designer might generate a concurrent design.

2.1.3 Methods for Producing Concurrent Designs

The following sections identify some notable methods for producing concurrent designs. Three categories of methods are discussed. Methods within the first category assume a RTSA model as a starting point but do not assume any particular target environment. Methods within the second category produce a concurrent design without assuming a RTSA model as a starting point. Methods within the third category assume that a concurrent design will execute within an Ada run-time environment.

2.1.3.1 Concurrent Designs from RTSA Models

Gomaa proposes several related methods for generating concurrent designs from a RTSA model of a problem. [Gomaa84, Gomaa93] One method, DARTS, includes a set of heuristics for mapping a data/control flow diagram into a set of concurrent tasks. Further heuristics address the problem of inter-task message communication and synchronization among tasks and shared modules. Once tasks and shared modules exist, DARTS relies on Structured Design [Yourdon79] to define the sequential processing within each task.

In later work, Gomaa describes a technique, called Concurrent Object-Based Real-time Analysis (COBRA), for adding objects to data/control flow diagrams. [Gomaa93] COBRA also provides a technique called behavioral-scenario analysis to help the analyst create an effective COBRA specification.

Gomaa augments his COBRA technique with an adaptation of DARTS, now called the COncurrent Design Approach for Real-Time Systems (CODARTS), that shows a designer how to create a concurrent design and a module structure from a COBRA specification. [Gomaa93]

2.1.3.2 Concurrent Designs without RTSA Models

Some methods for producing concurrent designs begin by producing a concurrent model of a problem and then map the model onto a concurrent design. One such method is Jackson System Development (JSD). [Jackson83] JSD identifies real-world entities within a problem and then models the behavior of those entities over time. JSD includes a notation, entity structure diagrams, for modeling entities and the sequence of events that occur in the life of each entity. Each entity can be viewed as an independent task that receives, and acts upon, events from the external environment. Once an entity model exists, JSD provides for construction of a network connecting the entities. Entities can be connected with either data streams or state vectors.

Another concurrent design approach that builds upon JSD is Entity-Life Modeling (ELM) proposed by Sanden. [Sanden94, Sanden89, Sanden89a] ELM models a problem with two basic components: entities with life (called threads) and resources (called objects). While not strictly limited to applications where Ada will be used, ELM results in a natural implementation of threads as Ada tasks and objects as Ada modules. Threads are identified by analyzing a problem in search of entities that have life. Objects are uncovered by looking for resources that are used by the entities. ELM avoids the need for heuristics to map from analysis to a concurrent design because the analysis itself produces a concurrent specification. In addition, Sanden believes that ELM results in fewer processes when compared with alternative approaches.

2.1.3.3 Concurrent Designs for Ada

Several concurrent design methods address specifically the creation of concurrent designs for implementation with Ada. Nielsen and Shumate propose one approach for designing real-time systems with Ada, given a data flow diagram model of a problem. [Nieslen88, Nielsen87] Nielsen and Shumate make two, restricting assumptions: 1) the system specification uses only data flow diagrams (that is, uses structured analysis) and 2) the target programming language is Ada. These assumptions affect the number and form of heuristics, as recommended by Nielsen and Shumate, needed to transform a data flow diagram into a concurrent design. Overall, the heuristics proposed by Nielsen and Shumate can be considered a subset of the more comprehensive heuristics recommended by Gomaa.

Another method, due to Buhr, uses Ada as a system design language and provides a pictoral representation for the semantics that can be expressed using Ada. [Buhr84] Buhr intends that a designer think about the problem using the semantic concepts provided by Ada, but augmented with a symbol, so-called "cloud", for concepts that do not exist within Ada. Perhaps the largest contribution of this approach is the notation itself, which is used within a number of other methods, for example ELM and the Nielsen and Shumate method, to provide a pictoral representation of the Ada mapping for a concurrent design.

2.1.4 Methods for Producing Object-Oriented Designs

While the advent of Ada and growth in real-time applications encouraged research into analysis and design methods for concurrent software, research into abstract-data-type theory, and the evolution of related, programming languages, spurred the development of analysis and design approaches oriented around objects. [Booch91, Booch86, Coad92, Coad91, Rumbaugh91, Shlaer92] In general, these approaches model a problem as a set of objects and relationships among them. The relationships include, prominently, generalization/specialization, aggregation, and client/server. In addition, most of these approaches provide models for specifying inter-object message passing and intra-object behavior.

2.1.5 Approaches to Design Scheduling

No matter what method a designer uses to generate a design for real-time software, performance requirements must be satisfied. In general, however, two distinct approaches exist to ensure that a design satisfies performance goals: deterministic approaches, requiring sequential designs, and concurrent approaches. Deterministic approaches are most often used and have at least a thirty-year history. [Peng93, Shepard91, Xu93] Concurrent approaches, gaining in popularity, have only about a ten-year history of use in real-time applications. [Obenza93, SEI92, Sha90] The community of researchers and practitioners of real-time design methods remains divided on which class of methods achieves the best results. Supporters of deterministic approaches argue that concurrent designs cannot ensure that application timing constraints will be met. Supporters of concurrent designs argue that deterministic approaches result in designs that are difficult to understand and maintain. Further, advocates of concurrent approaches believe that recent results in the area of rate monotonic scheduling theory can be used to ensure that concurrent designs will meet application timing constraints.

2.2 Automating Software Design Methods

Designing software requires that a designer possess both creativity and a capacity for complexity and detail. A number of researchers investigate automated approaches to assist designers with the intricacies of software design, without unduly restricting the creative aspects of the design process. Some researchers address design at the architectural level, while others consider assistance for detailed design. The following sections describe a number of approaches in each of these categories.

2.2.1 Approaches to Automating Architectural Design

Architectural design involves identifying the main elements of a design, defining the relationships among those elements, evaluating the resulting combination of elements and relationships, and then iterating these operations, as necessary, to consider alternate designs. Typically, this process requires substantial labor, leads to numerous errors, and proves difficult to evaluate, until much later in the software development process. The cost of producing and evaluating designs tends to be high, while the cost of producing inadequate designs, that go unverified until system testing, tends to be much higher. These factors motivate researchers to investigate automated methods to assist software architects.

2.2.1.1 Cluster Analysis

A system called Computer-Aided Process Organization, or CAPO, embodies one approach to transform a data flow diagram into a structured design. [Karimi88] CAPO strives to free a designer from using structured design techniques, such as transform and transaction analysis, to create structure charts. CAPO represents a data flow diagram as a flow graph, and then converts that flow graph into six matrices, used to compute an interdependency weight for the links joining each pair of transformations. Based upon the computed weights, CAPO converts the flow graph into a weighted, directed graph, and then uses a number of cluster analysis techniques to decompose that directed graph into a set of non-overlapping subgraphs.

CAPO provides no automated traceability between the flow graph and the resulting structure charts; such mapping must be determined by human inspection. CAPO also provides no automated assistance for checking the completeness and consistency of the proposed structure charts. In addition, CAPO does not capture the design rationale used to propose the various structure charts. In fact, wide variations in proposed structure charts can be obtained without changing the structure of the flow graphs by manipulating various numbers assigned to elements of the input flow graph. CAPO generates alternate designs by using various clustering algorithms but does not consider aspects of the target environment that might suggest alternate designs.

2.2.1.2 Formal Rule Rewriting Systems

Boloix, Sorenson, and Tremblay describe another approach, based on an entity-aggregate-relationship-attribute (EARA) model, to automatically transform data flow diagrams to structure charts. [Boloix92] Here, transformation rules, based on set theory, convert data flow diagrams, described formally at the lowest level of decomposition using an EARA model, into a formal description of structure charts. A human analyst then improves the resulting structure charts.

The EARA approach provides no automated completeness and consistency checking for the generated structure charts. In addition, the approach fails to capture the rationale used to generate the structure charts. Nor does the approach give consideration to generating alternate designs based upon variations in the intended run-time environment for the system under design. When consulting the designer at numerous points in the design-generation process, the EARA method does not vary the scope and nature of this elicitation based on the designer's level of experience.

2.2.1.3 Artificial Intelligence Approaches

A number of other proposals for automating software design can be classified as artificial intelligence, or AI, approaches because they apply techniques developed by AI researchers. These proposals can be divided into two subclasses: those based on rule-based expert systems and those based on more general, AI problem-solving paradigms.

2.2.1.3.1 Rule-based Expert Systems

Rule-based expert systems encode knowledge as production rules, where each rule takes the following form: **if antecedent then consequences**. Whenever a situation, or system state, represented as a set of facts, matches the antecedent of a rule, then that rule becomes eligible to execute. An inference engine cycles continuously through the set of facts by: 1) matching rules against facts, 2) selecting an eligible rule for execution, and 3) updating the facts set based on the consequences of the selected rule. This cycle continues until the rules known to the inference engine fail to match any facts known to the inference engine.

2.2.1.3.1.1 Specification-Transformation Expert System

Tsai and Ridge describe a Specification-Transformation Expert System (STES) that automatically translates a specification model (expressed as data flow diagrams) into a sequential design (expressed as structure charts). [Tsai88] The STES, implemented using the OPS5 expert-system shell, encapsulates the Structured Design method of Yourdon and Constantine within expert-system rules. [Yourdon79] STES represents both data flow diagrams and structure charts as structured facts. STES uses several textbook heuristics, including coupling, cohesion, fan-in, and fan-out, to guide the design process. Each data flow in a data flow diagram has an associated data dictionary entry that can be used by STES to gauge the degree of coupling between modules in a structure chart. An expert system has difficulty determining cohesion among functions, and so STES consults a user for information required to make inferences about functional cohesion. STES attempts to maximize fan-in and tries to achieve a moderate span of control.

STES operates as a sequential set of phases. First, STES factors the data flow diagram into afferent, efferent, and transform-centered branches. This factoring results in a top-level design for the structure chart. Second, STES refines each module at the next level of the structure chart using textbook guidelines for coupling, cohesion, fan-in, and fan-out. Third, STES renders the resulting, multilevel, structure chart using a CASE system from Cadre Technologies.

The approach embodied within STES limits its application to small designs, amenable to the sequential processing paradigm known as "inputs-processing-outputs".

In addition, the STES provides no automated checking for completeness and consistency of the generated structure chart. Traceability between the data flow diagram and the structure chart must be verified manually. STES does not capture the rationale for design decisions. Though consulting the designer at various times, STES does not temper the nature of such consultation based on the designer's level of experience. STES cannot generate alternate designs without changing the data flow diagram.

2.2.1.3.1.2 System Architects Apprentice, Design Assistant

Another approach, reported in the literature by Lor and Berry, transforms requirements into a design, but without using structure charts as the target. [Lor91] This semi-automated, knowledge-based approach, developed by Lor as the subject of a Ph.D. dissertation within the context of the System ARchitects Apprentice (SARA), a joint development of researchers at UCLA and the University of Wisconsin [Estrin86], builds on the SARA environment by providing automated assistance to help a designer transform a requirements specification into a SARA structural model and graph model of behavior, or GMB. Lor uses data flow diagrams and system verification diagrams to specify requirements. System verification diagrams provide a stimulus-response model of behavior that Lor uses to specify interactions among subsystems within a design. Lor uses data flow diagrams mainly to specify the interior of subsystems.

Lor chose a rule-based approach for his design assistant for two reasons. First, since the current set of rules for transforming requirements into SARA designs remains incomplete, locking the knowledge into a procedural program appears premature.

Second, the sequence of rule firings provides a natural explanation facility for design choices. The design assistant encompasses 21 rules for building the structural model, 59 for synthesizing the control domain, and 37 for modeling the data domain. Lor's approach synthesizes a SARA structural model through a direct translation of the hierarchy of data flow diagrams; at the lowest level of decomposition, the data flows map to SARA domain primitives. Lor's approach also creates a SARA GMB from the stimulus-response model provided by the system verification diagrams, as well as from the data flow diagrams.

Lor reports that his research provides a better understanding of, and a methodical approach to, designing systems within the SARA environment. The rules encapsulated in the design assistant can be called syntactically complete because every requirements construct is covered. The rules cannot, however, be called semantically complete; alternative designs cannot be considered and the rules cannot always map each requirements element to the most concise design construct. A human designer must answer queries as the design progresses (to provide needed information and to indicate preferences), and must improve the generated design. Given the same requirements specification technique (i.e., system verification diagrams and data flow diagrams), Lor asserts that his approach could be adapted to other design representations by rewriting the rule consequents; however, since the most crucial step in Lor's approach entails developing formal definitions, represented by SARA design constructs, for every construct in his requirements language, adapting to another design representation would require that this most crucial step be repeated.

2.2.1.3.2 Other Artificial Intelligence Approaches

Fickas' Critter, based on an earlier tool known as Glitter [Fickas90], provides an automated assistant that attempts to bridge between requirements and design for composite systems, those containing a mixture of human, hardware, and software components. [Fickas92] Critter uses an artificial intelligence paradigm of state-based search, relying on a human user to provide the domain knowledge necessary to guide the search. Critter encapsulates only domain-independent, design knowledge. Critter and a human designer interact to develop a design to solve a domain-specific problem. To date, the results with Critter appear unencouraging. Critter's limited reasoning techniques prevent its use on large software engineering problems; the analysis algorithms used in Critter prove too slow for an interactive design system; Critter's knowledge-base and representation omit several classes of system design concepts.

2.2.2 Approaches to Automating Detailed Design

A number of other researchers investigate methods of providing automated assistance for detailed design. Most such methods assume the existence of one or more architectural designs. The assistance then focuses on locating and modifying, or on creating, components that can be fitted into one of the existing architectures. In most cases, detail-design assistants operate in a narrow domain. [Fischer92, Rich88, Rich88a, Rich92

Waters91] Automated assistants of this type might be useful after the designer has a system architecture in place.

2.3 Automatic Programming

Unlike automated, design assistants, which help a human analyst complete a single, if essential, transformation in the software development process, automatic-programming systems attempt to perform, without human intervention, every transformation required to generate a working implementation from an initial specification of user requirements. [Barstow85, Barstow91, Kant91, Setliff91, Smith91] A different form of automatic programming, end-user programming, enables a computer-naive user to interact with an intelligent agent to select, exercise, evaluate, and modify an application program. End-user programming requires no formal specification of requirements; in fact, the user need only bring the ideas in his head to a computer terminal to begin the process. [Marques92]

Numerous problems block success with automatic programming. Many automatic programming research projects seem to be limited to a single, small domain. Even in such projects the number and type of transformations required to convert a moderate specification into a program can be enormous. In addition, the automatic generation process produces a huge repository of data that can be difficult to manage. Further, the knowledge contained within an automatic programming system is dispersed widely and, thus, modifying such a system can be challenging. When an automatic programming system produces incorrect results, end users tend to examine the target code for the cause of the errors. Such an approach to software debugging, reminiscent of programmers who would modify the object code produced by a faulty compiler, can be costly, risky, and unproductive. Experience to date indicates that automatic programming will remain confined, for the foreseeable future, to single, small application domains.

The end-user variant of automatic programming systems overcomes the limits of a single, small domain, but at a cost. End-user programming systems require that a user sort through a range of problem-solving strategies in an effort to determine which approach might best meet his problem. After selecting an approach, the user must interact with the program over a long period of time until the performance of the program meets the user's expectations. As one possible outcome of this prolonged interaction, the user or the system might realize that the initial problem-solving strategy was, in fact, wrong. The basic approach to end-user programming seems to be educated guess, followed by trial and error refinement. Few users have the patience for such an approach to programming.

2.4 Summary of Findings

A number of significant findings might be taken from the preceding review of the literature concerning software design methods, automated, design assistants, and automatic programming. Three findings merit particular attention.

2.4.1 Importance of Architectural Design

The most critical portion of the software design process occurs as a designer transforms the software requirements into an architectural design, consisting of design elements and the relationships among design elements. Up until an architectural design exists, the costs associated with producing a system are usually small. After an architectural design exists, the costs associated with producing a system increase substantially as detailed designers, coders, testers, and document writers start to work. In addition, the cost of correcting errors escalates substantially as more and more of the system exists. And, of course, errors made in the software architecture typically produce a wider effect than errors made during detailed design and coding.

The designer of the architecture for a software system bears a large responsibility. The designer must ensure the completeness, consistency, and correctness of requirements, both functional and nonfunctional. The designer must ensure a mapping from every element in the requirements specification to one or more element in the design. The designer establishes the intellectual model under which the remainder of the software development proceeds, including, most importantly, the software maintenance. The software architecture must be understandable, consistent, and complete. To the extent possible, a number of alternative architectures should be considered, and compared in terms of clarity, performance, and, cost.

2.4.2 Benefits of Software Design Methods

Software design methods exist to improve a human designer's ability to meet the challenges of architectural design in a rigorous, consistent, and repeatable fashion. Design methods also help a designer to identify what to do first, to recognize at any point what to do next and when to do it, and to establish when a suitable design exists. The notations associated with a design method improve the ability of the designer to communicate with other software developers.

2.4.3 Benefits from Automating Software Design Methods

Automation can benefit software design methods in several ways: improved rigor, increased effectiveness, and reduced variability. By encoding knowledge about what to do and when to do it, and about what information is needed and when it is needed, automation can ensure that a designer does not overlook any detail of a design method. In addition, design expertise can be accumulated incrementally, as the expertise is acquired and tested. An automated tool might handle routine design chores, where possible without involving a human designer. Yet, an appropriate approach might enable a human designer to review and understand all decisions made by such a tool.

Automated representations of design products can help get precise versions of the requirements and design on paper early, and in a form that can provide automatic traceability between requirements and design and that can permit the design to be checked for completeness and consistency among all its forms. Once a design exists in an automated form, verification, simulation, and analysis of the design becomes much easier. An automated form of a design also increases the likelihood of reusing part or all of a design and facilitates the generation of code skeletons for tasks and modules. In addition, given a suitable internal representation, ad hoc questions about a design can be answered and the relationships among design elements can be tracked, with the assistance of an automated query system.

Appropriate automated support might also improve the utility of computer-aided software engineering (CASE) tools. At the present state of development, most CASE tools allow a designer to construct various diagrammatic representations, such as data flow diagrams and structure charts. These diagrams, kept in an internal form, can be checked to some degree for self-consistency and for completeness. In most cases, a designer must convert one type of a diagram, say a data flow diagram, to another form, say a structure chart, using manual methods outside the CASE tool. Existence of appropriate automation would permit the designer to generate automatically, within the CASE tool, one diagram from the other.